(http://twitter.com/kenlistdev) (http://github.com/kenlist)

 (http://geekluo.com/)

(mailto:rijian.lrj@alibaba-inc.com)

Lua数据结构 -- TString（二）

2014/04/11

存储lua里面的字符串的**TString数据结构**:（lobject.h 196-207）

```
/*
** String headers for string table
*/
typedef union TString {
  L_Umaxalign dummy;  /* ensures maximum alignment for strings */
  struct {
    CommonHeader;
    lu_byte reserved;
    unsigned int hash;
    size_t len;
  } tsv;
} TString;
```

其它结构中也会有**L_Umaxalign dummy**这个东西，来看看L_Umaxaliagn:

```
#define LUAI_USER_ALIGNMENT_T union { double u; void *s; long l; }
typedef LUAI_USER_ALIGNMENT_T L_Umaxalign;
```

从字面意思上就是保证**内存能与最大长度的类型进行对齐**，事实上也是做这件事，这里感觉lua想给各种不同设备做一种嵌入式脚本，这里要保证与最大的长度对齐能**保证CPU运行高效不会罢工。**

tsv才是TString的主要数据结构：

1. CommonHeader：这个是和GCObject能对应起来的GCheader
2. reserved：保留位
3. hash：每个字符串在创建的时候都会用有冲突的哈希算法获取哈希值以提高性能
4. len：字符串长度

哈希是lua里一个很重要的优化手段，具体的哈希算法相关知识在文章最后会补充说明一下，字符串的hast表放在L->l_G->strt中，这个成员的类型是stringtable，我们再来看看**stringtable数据结构**：（lstate.h 38-42）

```
typedef struct stringtable {
  GCObject **hash;
  lu_int32 nuse;  /* number of elements */
  int size;
} stringtable;
```

stringtable结构很简单：

1. hash：一个GCObject的表，在这里其实是个TString*数组

2. nuse：已经有的TString个数
3. size：hash表的大小(可动态扩充)

接下来看看stringtable是怎么动态调整大小的: ####1 动态扩充stringtable：(lstring.c 60-70)

```
if (tb->nuse > cast(lu_int32, tb->size) && tb->size <= MAX_INT/2)
        luaS_resize(L, tb->size*2);  /* too crowded */
```

每次newlstr的之后，都会判断nuse是否已经大于table的size，如果是的话就会重新resize这个stringtable的大小为原来的2倍。

####2 动态浓缩stringtable：(lgc.c 433-436)

```
  if (g->strt.nuse < cast(lu_int32, g->strt.size/4) &&
      g->strt.size > MINSTRTABSIZE*2)
    luaS_resize(L, g->strt.size/2);  /* table is too big */
```

在gc的时候，会判断nuse是否比size/4还小，如果是的话就重新resize这个stringtable的大小为原来的1/2倍。

####3 resize算法：(lstring.c 22-47)

```
  newhash = luaM_newvector(L, newsize, GCObject *);
  /* rehash */
  for (i=0; i<tb->size; i++) {
    GCObject *p = tb->hash[i];
    while (p) {  /* for each node in the list */
      GCObject *next = p->gch.next;  /* save next */
      unsigned int h = gco2ts(p)->hash;
      int h1 = lmod(h, newsize);  /* new position */
      lua_assert(cast_int(h%newsize) == lmod(h, newsize));
      p->gch.next = newhash[h1];  /* chain it */
      newhash[h1] = p;
      p = next;
    }
  }
```

resize时，需要根据每个节点的哈希值重新计算新位置，然后放到newhash里。

字符串在哪里? 看完TString和stringtable，大家都没有发现究竟字符串放在哪里，从内存上看其实字符串直接放在了**TString后面**，这样还能省掉一个成员：(lstring.c 56)

```
TString { ··· } | 'abcdefg/0'
ts = cast(TString *, luaM_malloc(L, (l+1)*sizeof(char)+sizeof(TString)));
```

###性能问题： 在这里说一点lua的性能问题，虽然不在这个主题的讨论范围。由上面可以知道lua的字符串是带hash值的，所以我们拿着一个字符串去做**比较、查询、传递**等操作都是**非常高效**的。

但是我们也可以看到**每次创建一个新的字符串**都会做很多操作，所以这里**不建议频繁做字符串创建、连接、销毁等操作**，最好能**缓存**一下。

###补充：字符串的哈希算法： 常用的字符串哈希函数比较如下：

Hash 函数	数据 1	数据 2	数据 3	数据 4	数据 1 得分	数据 2 得分	数据 3 得分	数据 4 得分	平均分
BKDRHash	2	0	4774	481	96.55	100	90.95	82.05	92.64
APHash	2	3	4754	493	96.55	88.46	100	51.28	86.28
DJBHash	2	2	4975	474	96.55	92.31	0	100	83.43
JSHash	1	4	4761	506	100	84.62	96.83	17.95	81.94
RSHash	1	0	4861	505	100	100	51.58	20.51	75.96
SDBMHash	3	2	4849	504	93.1	92.31	57.01	23.08	72.41
PJWHash	30	26	4878	513	0	0	43.89	0	21.95
ELFHash	30	26	4878	513	0	0	43.89	0	21.95

其中数据1为100000个字母和数字组成的随机串哈希冲突个数。数据2为100000个有意义的英文句子哈希冲突个数。数据3为数据1的哈希值与1000003(大素数)求模后存储到线性表中冲突的个数。数据4为数据1的哈希值与10000019(更大素数)求模后存储到线性表中冲突的个数。

经过比较，得出以上平均得分。平均数为平方平均数。可以发现，BKDRHash无论是在实际效果还是编码实现中，效果都是最突出的。APHash也是较为优秀的算法。DJBHash,JSHash,RSHash与SDBMHash各有千秋。PJWHash与ELFHash效果最差，但得分相似，其算法本质是相似的。

####在Lua中使用到的是JSHash算法：

```
// JS Hash
Unsigned int JSHash(char *str)
{
    Unsigned int hash = strlen(str); //使用字符串长度作为seed
    While (*str)
    {
        hash ^= ((hash << 5) + (*str++) + (hash >> 2));
    }
    return hash;
}
```

具体JS Hash算法的冲突性解决和性能上面，我也不懂，具体要找paper看看，但是从数据比较上看，JSHash是属于较好的算法，可也有比JSHash算法更好的字符串哈希算法。

lua (6) (http://geekluo.com/tags.html#lua-ref)

← Previous (http://geekluo.com/contents/2014/04/11/3-lua-table-structure.html)

Next → (http://geekluo.com/contents/2014/04/12/5-lua-closure-structure.html)